PPCG and Pencil Compiler Design

Sven Verdoolaege

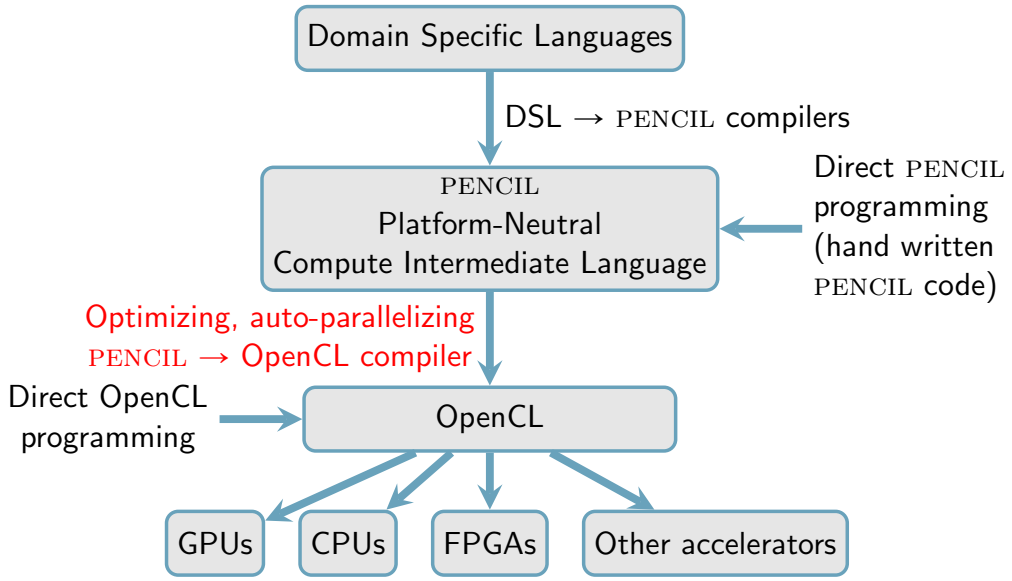
Sven.Verdoolaege@gmail.com

May 11, 2016

CARP Project (2011–2015)

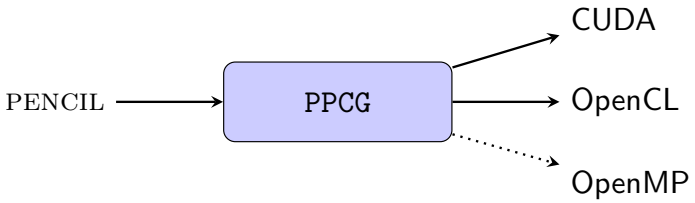
Design tools and techniques to aid

Correct and Efficient Accelerator Programming



PPCG Overview

[12]



PPCG:

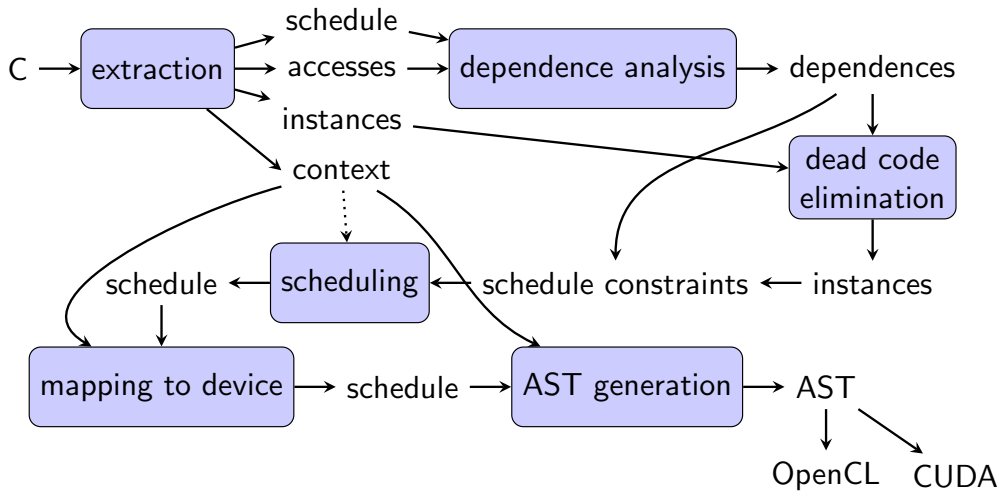
- detect/expose parallelism
- map parts of the code to an accelerator
- copy data to/from device
- introduce local copies of data

PENCIL:

- C99 with restrictions and some extra builtins and pragmas PENCIL

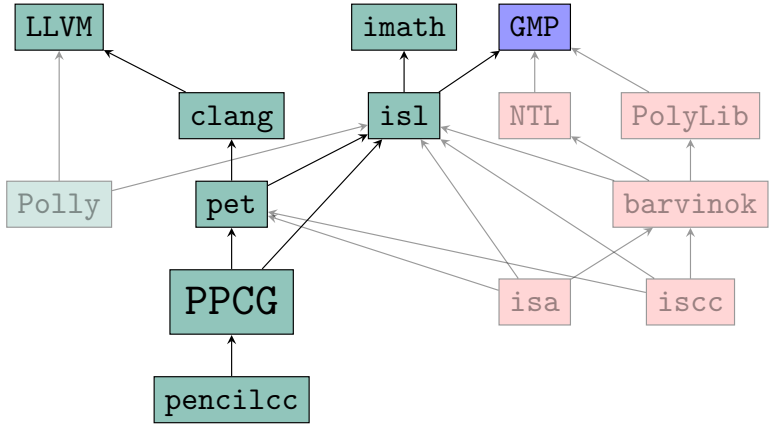
PPCG Internal Structure

[12]



Note: as currently implemented (version 0.06), not necessarily how it should be implemented

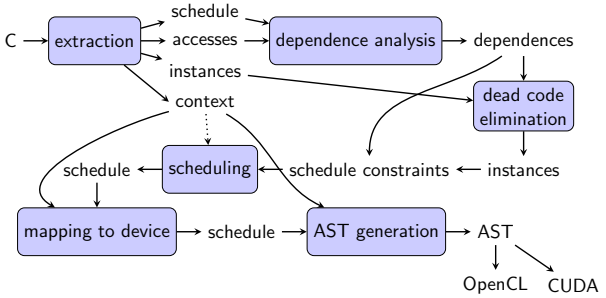
Connection with other Libraries and Tools



Licenses:
BSD/MIT
LGPL
GPL

`isl`: manipulates parametric affine sets and relations
`pet`: extracts polyhedral model from clang AST
`PPCG`: Polyhedral Parallel Code Generator
`pencilcc`: PENCIL compiler

Outline



Polyhedral Model [8]

Main constituents of program representation

- Instance Set
  - ⇒ the set of all statement instances
- Access Relations
  - ⇒ the array elements accessed by a statement instance
- Dependences
  - ⇒ the statement instances that depend on a statement instance
- Schedule
  - ⇒ the relative execution order of statement instances
- Context
  - ⇒ constraints on parameters

For extracting a polyhedral model from C, `PPCG` uses

- `pet` for instance set, access relations, initial schedule and context
- `isl` for computing dependences

Polyhedral Model Requirements

Requirements for `PPCG`:

- Sufficient static control
  - ⇒ static control (not depending on input data) is represented directly
  - ⇒ internal (structured) dynamic control is encapsulated
- Affine
  - ⇒ all relevant expressions are (quasi-)affine, or
  - ⇒ all relevant expressions can be approximated as (quasi-)affine
- No Aliasing
  - ⇒ essentially no pointer manipulations

Parametric Example: Matrix Multiplication

```
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
S1:   C[i][j] = 0;
      for (int k = 0; k < K; k++)
S2:       C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```

- Instance Set (set of statement instances)

$$\{ \mathtt{S1}[i,j] : 0 \leqslant i < M \wedge 0 \leqslant j < N; \\ \mathtt{S2}[i,j,k] : 0 \leqslant i < M \wedge 0 \leqslant j < N \wedge 0 \leqslant k < K \}$$

- Access Relations (accessed array elements; W: write, R: read)

$$W = \{ \mathtt{S1}[i,j] \to \mathtt{C}[i,j]; \mathtt{S2}[i,j,k] \to \mathtt{C}[i,j] \}$$
$$R = \{ \mathtt{S2}[i,j,k] \to \mathtt{C}[i,j]; \mathtt{S2}[i,j,k] \to \mathtt{A}[i,k]; \mathtt{S2}[i,j,k] \to \mathtt{B}[k,j] \}$$

Schedule Representation

[11]

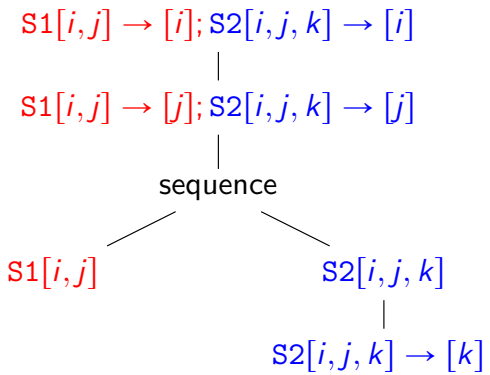
Schedule S keeps track of relative execution order of statement instances

⇒ for each pair of statement instances **i** and **j**, schedule determines
- **i** executed before **j**
- **i** executed after **j**
- **i** and **j** may be executed simultaneously

In PPCG, schedules are represented as schedule trees
- Main node types
  - *band*: instances are executed according to associated multi-dimensional piecewise quasi-affine partial schedule
    the elements of a band are called its *members*
  - *sequence*: children are executed in order
- Deriving schedule tree from AST
  - `for` loop ⇒ single-dimensional band
  - compound statement ⇒ sequence

Parametric Example: Matrix Multiplication

```
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
S1:   C[i][j] = 0;
      for (int k = 0; k < K; k++)
S2:       C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```



Instance Set

Region that needs to be extracted may be
- marked by

```
#pragma scop
#pragma endscop
```

- autodetected (`--pet-autodetect`)

Internal structured dynamic control is encapsulated

```
for (int x = 0; x < n; ++x) {
A:      s = f();
B:      while (P(x, s))
            s = g(s);
C:      h(s);
}
```

Instance set: $\{ \mathtt{A}[x] : 0 \leqslant x < n; \mathtt{B}[x] : 0 \leqslant x < n; \mathtt{C}[x] : 0 \leqslant x < n \}$

Note: currently, internal order of accesses is lost
⇒ possible loss of accuracy in dependence analysis

Inlining

- currently, only supported for outermost call expression
- enable through C99 `inline` keyword on function definition

```
inline void set_diagonal(int n,
        float A[const restrict static n][n], float v)
{
        for (int i = 0; i < n; ++i)
U:              A[i][i] = v;
}

void f(int n, float A[const restrict static n][n])
{
#pragma scop
S:      set_diagonal(n, A, 0.f);
        for (int i = 0; i < n; ++i)
                for (int j = i + 1; j < n; ++j)
T:                      A[i][j] += A[i][j - 1] + 1;
#pragma endscop
}
```

Instance set: $\{ \mathtt{U}[i] : 0 \leqslant i < n; \mathtt{T}[i,j] : 0 \leqslant i < j < n \}$

Access Relations

```
for (int x = 0; x < n; ++x) {
A:      s = f();
B:      while (P(x, s))
                s = g(s);
C:      h(s);
}
```

Three types of access relations:

- May-read: $\{ \mathtt{B}[x] \to \mathtt{s}[] : 0 \leqslant x < n; \mathtt{C}[x] \to \mathtt{s}[] : 0 \leqslant x < n \}$
- May-write : $\{ \mathtt{A}[x] \to \mathtt{s}[] : 0 \leqslant x < n; \mathtt{B}[x] \to \mathtt{s}[] : 0 \leqslant x < n \}$
- Must-write : $\{ \mathtt{A}[x] \to \mathtt{s}[] : 0 \leqslant x < n \}$

Access Relations and Function Calls

```
void set_diagonal(int n,
        float A[const restrict static n][n], float v)
{
        for (int i = 0; i < n; ++i)
U:              A[i][i] = v;
}

void f(int n, float A[const restrict static n][n])
{
#pragma scop
S:      set_diagonal(n, A, 0.f);
        for (int i = 0; i < n; ++i)
                for (int j = i + 1; j < n; ++j)
T:                      A[i][j] += A[i][j - 1] + 1;
#pragma endscop
}
```

May-write: $\{ \mathtt{S}[] \to \mathtt{A}[i,i] : 0 \leqslant i < n; \mathtt{T}[i,j] \to \mathtt{A}[i,j] : 0 \leqslant i < j < n \}$
Must-write: $\{ \mathtt{S}[] \to \mathtt{A}[i,i] : 0 \leqslant i < n; \mathtt{T}[i,j] \to \mathtt{A}[i,j] : 0 \leqslant i < j < n \}$

Access Relations and Structures

[7]

```
struct s {
        int a;
        int b;
};

int f()
{
        struct s a, b[10];

S:      a.b = 57;
T:      a.a = 42;
        for (int i = 0; i < 10; ++i)
U:              b[i] = a;
}
```

Must-write: $\{ \mathtt{S}[] \to \mathtt{a\_b}[\mathtt{a}[] \to \mathtt{b}[]]; \mathtt{T}[] \to \mathtt{a\_a}[\mathtt{a}[] \to \mathtt{a}[]]; \mathtt{U}[i] \to \mathtt{b\_a}[\mathtt{b}[i] \to \mathtt{a}[]]; \mathtt{U}[i] \to \mathtt{b\_b}[\mathtt{b}[i] \to \mathtt{b}[]] \}$

Summary Functions

[1, 7]

Analysis of accesses in called function may be inaccurate or even infeasible

- missing body (library function without source)
- unstructured control
- aliasing
- pattern inside dynamic control is ignored
- additional information not explicitly expressed in code

⇒ explicitly specify accesses in summary function PENCIL

Summary Function Example

```
int f(int i); int maybe(); struct s { int a; };
void set_odd_summary(int n, struct s A[static n]) {
        for (int i = 1; i < n; i += 2)
                if (maybe())
                        A[i].a = 0;
}
__attribute__((pencil_access(set_odd_summary)))
void set_odd(int n, struct s A[static n])
{
        for (int i = 0; i < n; ++i)
                A[2 * f(i) + 1].a = i;
}
void foo(int n, struct s B[static 2 * n])
{
#pragma scop
S:      set_odd(2 * n, B);
#pragma endscop
}
```

May-write: $\{ S[] \to B\_a[B[i] \to a[]] : 0 \leqslant i < 2n \wedge i \bmod 2 = 1 \}$

Context

The context collects constraints on the symbolic constants

- derived by `pet`
  - exclude values that result in undefined behavior
    - ⋆ negative array sizes
    - ⋆ out-of-bounds accesses
    - ⋆ signed integer overflow
  - `__builtin_assume` or `__pencil_assume` PENCIL
    - ⇒ any constraint can be specified
    - ⇒ only quasi-affine constraints on symbolic constants are exploited
- specified on `PPCG` command line
  - `--ctx`
  - `--assume-non-negative-parameters`

Main purpose: simplify generated AST

Dependence analysis in `isl`

[8, 9]

`isl` contains generic dependence analysis engine

⇒ determines dependence relations between "sources" and "sinks"

Input:

- Sink $K : I \to D$
- May-source $Y : I \to D$
- Must-source $T : I \to D$
- Schedule S on I ⇒ defines "before" and "intermediate"

Output:

- May-dependence relation: triples $(\mathbf{i}, \mathbf{k}, \mathbf{a})$
  - **k** has a sink to **a**
  - **i** has a may or must source to **a** before **k**
  - there is no intermediate must source to **a**
- Must-dependence relation: triples $(\mathbf{i}, \mathbf{k}, \mathbf{a})$
  - **k** has a sink to **a**
  - **i** has a must source to **a** before **k**
  - there is no intermediate may or must source to **a**
- May-no-source: sinks $\mathbf{k} \to \mathbf{a}$ with no must source to **a** before **k**
- Must-no-source: sinks $\mathbf{k} \to \mathbf{a}$ with no may or must source to **a** before **k**

Dependence analysis in PPCG [9]

isl:

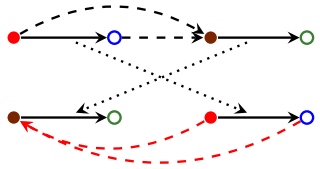
- May-dependence relation: triples $(\mathbf{i}, \mathbf{k}, \mathbf{a})$
  - $\mathbf{k}$ has a sink to $\mathbf{a}$
  - $\mathbf{i}$ has a may or must source to $\mathbf{a}$ before $\mathbf{k}$
  - there is no intermediate must source to $\mathbf{a}$
- May-no-source: sinks $\mathbf{k} \rightarrow \mathbf{a}$ with no must source to $\mathbf{a}$ before $\mathbf{k}$

PPCG (without live-range reordering):

- flow dependences (without $\mathbf{a}$) and live-in (may-no-source)
  - sink: may-read
  - may-source: may-write
  - must-source: must-write
- false dependences (without $\mathbf{a}$)
  - sink: may-write
  - may-source: may-read or may-write
  - must-source: must-write
- killed writes (without $\mathbf{k}$) ($\Rightarrow$ removed from may-write to get live-out)
  - sink: must-write
  - may-source: may-write

Live-Range Reordering [7, 9]

```
a = f1();
f2(a);
a = f3();
f4(a);
```



⟶: flow

- - ➝: false

Reordering rejected due to false dependences

Live-range reordering

- allows such live-ranges to be reordered
- using somewhat different classification of dependences
- computed using different calls to the same dependence analysis engine

Kills [7]

Basic idea:

- Must-writes kill dependences to earlier writes
- Pure kills can also be useful
- Treated as must-writes during dependence analysis, but
- Removed from dependence relations

Kills can be inserted

- automatically by pet
  - Variable declared within SCoP
    - $\Rightarrow$ kill at declaration
    - $\Rightarrow$ kill at end of enclosing block (if within SCoP)
  - Variable declared in scope that contains SCoP, only used inside
    - $\Rightarrow$ kill at end of SCoP
- manually by the user
  - `__pencil_kill` PENCIL

Dependence analysis in PPCG [9]

isl:

- May-dependence relation: triples $(\mathbf{i}, \mathbf{k}, \mathbf{a})$
  - $\mathbf{k}$ has a sink to $\mathbf{a}$
  - $\mathbf{i}$ has a may or must source to $\mathbf{a}$ before $\mathbf{k}$
  - there is no intermediate must source to $\mathbf{a}$
- May-no-source: sinks $\mathbf{k} \rightarrow \mathbf{a}$ with no must source to $\mathbf{a}$ before $\mathbf{k}$

PPCG (without live-range reordering):

- flow dependences (without $\mathbf{a}$) and live-in (may-no-source)
  - sink: may-read
  - may-source: may-write
  - must-source: must-write or kill
- false dependences (without $\mathbf{a}$)
  - sink: may-write
  - may-source: may-read or may-write
  - must-source: must-write
- killed writes (without $\mathbf{k}$) ($\Rightarrow$ removed from may-write to get live-out)
  - sink: must-write or kill
  - may-source: may-write

Kill Example

```
void f(int n, int A[restrict static n],
       int B[restrict static n])
{
       int t;
#pragma scop
       for (int i = 0; i < n; ++i) {
              t = A[i];
              B[i] = t;
       }
       __pencil_kill(t);
#pragma endscop
}
```

Without kill of t, compiler needs to assume t may be used after loop

⇒ last write needs to remain last

⇒ limited scheduling freedom (even with live-range reordering)

Note: kill inserted automatically by pet (if t not used after SCoP)

Absence of Loop Carried Dependences [7]

```
void foo(int n, int A[restrict static n][n],
         int B[restrict static n][n])
{
       for (int i = 0; i < n; ++i)
              #pragma pencil independent
              for (int j = 0; j < n; ++j)
                     B[i][A[i][j]] = i + j;
}
```

Assume each row of A has distinct elements

⇒ no loop-carried dependences, but PPCG cannot tell

⇒ add #pragma pencil independent PENCIL

Note: not handled very efficiently in current version of PPCG

⇒ only add when needed

Dead Code Elimination [7]

Basic idea:

1. Take statement instances that perform live-out access
   ⇒ "live instances"
2. Apply (reverse) dataflow dependence relation to live instances
   ⇒ (possibly) extra live instances
3. Repeat until no more extra live instances are found
4. Replace original instances by live instances

Naive implementation may not terminate

⇒ replace live instance set by its integer affine hull after each extension

⇒ bounded number of extensions

Dead Code Elimination Example

reg_detect from PolyBench 3.2

```
for (t = 0; t < _PB_NITER; t++)
  {
    for (j = 0; j <= _PB_MAXGRID - 1; j++)
      for (i = j; i <= _PB_MAXGRID - 1; i++)
        for (cnt = 0; cnt <= _PB_LENGTH - 1; cnt++)
          diff[j][i][cnt] = sum_tang[j][i];
    for (j = 0; j <= _PB_MAXGRID - 1; j++)
      {
        for (i = j; i <= _PB_MAXGRID - 1; i++)
          {
            sum_diff[j][i][0] = diff[j][i][0];
            for (cnt = 1; cnt <= _PB_LENGTH - 1; cnt++)
              sum_diff[j][i][cnt] = sum_diff[j][i][cnt - 1] +
                                    diff[j][i][cnt];
            mean[j][i] = sum_diff[j][i][_PB_LENGTH - 1];
          }
      }
    for (i = 0; i <= _PB_MAXGRID - 1; i++)
      path[0][i] = mean[0][i];
    for (j = 1; j <= _PB_MAXGRID - 1; j++)
      for (i = j; i <= _PB_MAXGRID - 1; i++)
        path[j][i] = path[j - 1][i - 1] + mean[j][i];
  }
```

Optimization Criteria for PPCG

- Two levels of parallelism
  - ⇒ blocks and threads (work groups and work items)
  - ⇒ parallelism

  In PPCG, second level obtained through tiling
  - ⇒ tilability
- Reduced working set for some arrays
  - ⇒ mapping to shared memory or registers

  Obtained through tiling
  - ⇒ tilability
- Reduced data movement
  - ⇒ locality
- Simple schedules
  - ⇒ schedule used in several subsequent steps, including AST generation
  - ⇒ simplicity

Scheduling Constraints

[9]

- Validity $\mathbf{a} \to \mathbf{b}$
  - ⇒ statement instance $\mathbf{b}$ needs to be executed after $\mathbf{a}$
  - ⇒ $f(\mathbf{b}) \geqslant f(\mathbf{a})$
- Proximity $\mathbf{a} \to \mathbf{b}$
  - ⇒ statement instance $\mathbf{b}$ preferably executed close to $\mathbf{a}$
  - ⇒ $f(\mathbf{b}) - f(\mathbf{a})$ as small as possible
- Coincidence $\mathbf{a} \to \mathbf{b}$
  - ⇒ statement instance $\mathbf{b}$ preferably executed together with $\mathbf{a}$
  - ⇒ $f(\mathbf{b}) = f(\mathbf{a})$
  - ⇒ band member only considered "coincident" if it coschedules all pairs
- Conditional validity (live-range reordering)
  - condition $\mathbf{b} \to \mathbf{c}$ (⇜ flow dependences)
  - conditioned validity $\mathbf{a} \to \mathbf{b}$, $\mathbf{c} \to \mathbf{d}$ (⇜ order dependences)

Schedule constraints only relevant if coscheduled by outer nodes
Other schedule constraints are said to be *carried* by some outer node

Dependences and Schedule Constraints

[9]

Traditional dependences

- flow dependences
  - ⇒ validity constraints
  - ⇒ proximity constraints
  - ⇒ coincidence constraints (when parallelism is important)
- false dependences
  - ⇒ validity constraints
  - ⇒ coincidence constraints (when parallelism is important)
  - ⇒ proximity constraints (optional for memory reuse)
- pairs of reads with shared write ("input dependences")
  - ⇒ proximity constraints (optional)

Live-range reordering

- somewhat different classification of dependences
- slightly different mapping to schedule constraints

Current PPCG

- adds false dependences to proximity constraints for historical reasons
- does not consider input dependences
- uses live-range reordering by default

Schedule Output

[11]

Scheduler produces a schedule tree

Main node types

- *band*: instances are executed according to associated multi-dimensional piecewise quasi-affine partial schedule

  the elements of a band are called its *members*

  some of the members are marked *coincident*
- *sequence*: children are executed in order
- *set*: children may be executed in any order

Scheduling Algorithms [2, 3]

Optimization criteria:

- parallelism
- tilability
- locality
- simplicity

Some well-known scheduling algorithms:

- Feautrier
  - carry as many (groups of) dependences as possible
  - fine-grained parallelism
- Pluto-algorithm
  - tilability (through permutability)
  - locality with parallelism as extreme case

PPCG uses variant of Pluto-algorithm with Feautrier fallback

⇒ force outer coincidence in each band

⇒ locally fall back to Feautrier if infeasible (single step)

Forced Outer Coincidence

PPCG uses variant of Pluto-algorithm with Feautrier fallback

⇒ force outer coincidence in each band

⇒ locally fall back to Feautrier if infeasible (single step)

Members in bands constructed by Pluto-algorithm are permutable

⇒ if outer member cannot be coincident, then no member can be

Each step in Feautrier algorithm carries as many dependences as possible

⇒ subsequent application of Pluto may find coincident member

Alternative: Pluto-algorithm + wavefront

all n members valid within outer nodes: $f_i(\mathbf{b}) \geqslant f_i(\mathbf{a})$

⇒ sum $\sum_i f_i$ is also valid

⇒ any schedule constraint carried by some f_i is also carried by sum

⇒ split band into two bands

1. outer band with single member corresponding to sum
2. inner coincident band with $n-1$ out of the n original members

But: sum may have large coefficients

⇒ not as simple as result of Feautrier

Outer Coincidence Example

jacobi-2d from PolyBench 4.1

```
   for (t = 0; t < _PB_TSTEPS; t++) {
       for (i = 1; i < _PB_N - 1; i++)
         for (j = 1; j < _PB_N - 1; j++)
S:         B[i][j] = SCALAR_VAL(0.2) * (A[i][j] + A[i][j-1] +
                                        A[i][1+j] + A[1+i][j] + A[i-1][j]);
       for (i = 1; i < _PB_N - 1; i++)
         for (j = 1; j < _PB_N - 1; j++)
T:         A[i][j] = SCALAR_VAL(0.2) * (B[i][j] + B[i][j-1] +
                                        B[i][1+j] + B[1+i][j] + B[i-1][j]);
    }
```

Pluto + wavefront:

$\mathtt{S}[t,i,j] \to t; \mathtt{T}[t,i,j] \to t$

$\mathtt{S}[t,i,j] \to 2t+i; \mathtt{T}[t,i,j] \to 2t+i+1$

$\mathtt{S}[t,i,j] \to 2t+i+j; \mathtt{T}[t,i,j] \to 2t+i+j+1$

Scheduling Algorithm May 11, 2016 39 / 66

Outer Coincidence Example

jacobi-2d from PolyBench 4.1

```
   for (t = 0; t < _PB_TSTEPS; t++) {
       for (i = 1; i < _PB_N - 1; i++)
         for (j = 1; j < _PB_N - 1; j++)
S:         B[i][j] = SCALAR_VAL(0.2) * (A[i][j] + A[i][j-1] +
                                        A[i][1+j] + A[1+i][j] + A[i-1][j]);
       for (i = 1; i < _PB_N - 1; i++)
         for (j = 1; j < _PB_N - 1; j++)
T:         A[i][j] = SCALAR_VAL(0.2) * (B[i][j] + B[i][j-1] +
                                        B[i][1+j] + B[1+i][j] + B[i-1][j]);
    }
```

Pluto + wavefront:

$\mathtt{S}[t,i,j] \to 5t+2i+j; \mathtt{T}[t,i,j] \to 5t+2i+j+2$

$\mathtt{S}[t,i,j] \to 2t+i; \mathtt{T}[t,i,j] \to 2t+i+1$

$\mathtt{S}[t,i,j] \to 2t+i+j; \mathtt{T}[t,i,j] \to 2t+i+j+1$

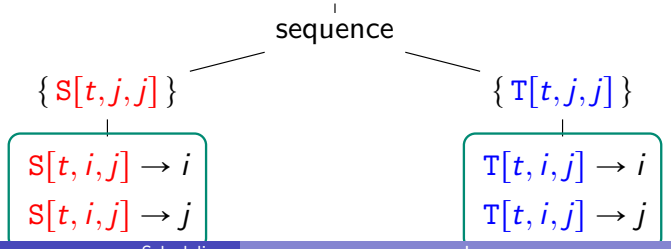
Outer Coincidence Example

jacobi-2d from PolyBench 4.1

```
   for (t = 0; t < _PB_TSTEPS; t++) {
       for (i = 1; i < _PB_N - 1; i++)
         for (j = 1; j < _PB_N - 1; j++)
S:         B[i][j] = SCALAR_VAL(0.2) * (A[i][j] + A[i][j-1] +
                                        A[i][1+j] + A[1+i][j] + A[i-1][j]);
       for (i = 1; i < _PB_N - 1; i++)
         for (j = 1; j < _PB_N - 1; j++)
T:         A[i][j] = SCALAR_VAL(0.2) * (B[i][j] + B[i][j-1] +
                                        B[i][1+j] + B[1+i][j] + B[i-1][j]);
    }
```

Feautrier + Pluto:

$$S[t,i,j] \to 2t; T[t,i,j] \to 2t+1$$

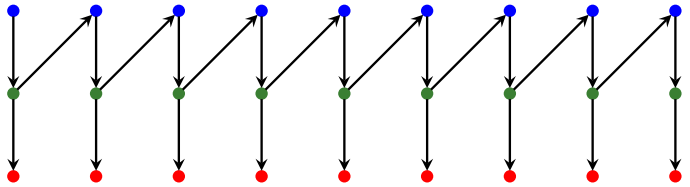


Known Issues with Scheduling Algorithms

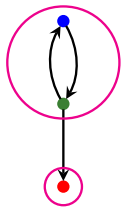
- Scheduling may take a long time
- Schedule may result in loop coalescing
- Proximity constraints may affect feasibility (Pluto)
- Schedule may be unnecessarily scaled (Feautrier)

Strongly Connected Components

Dependence graph (statement instances)



Dependence graph (statements)



Strongly connected components

Long Scheduling Times

Problem

- Feautrier requires solution of (large) LP problem
- Pluto requires solution of (large) ILP problem

Possible mitigations

- Group closely related statements into a single statement
  - ⇒ `--group-chains` (enabled by default in `PPCG`)
- Perform scheduling incrementally (Pluto-algorithm)
  1. First schedule SCCs separately
  2. Then combine SCCs incrementally

  - ⇒ better control over coincidence and band depth
    - ⇒ refuse combination if it reduces coincidence or band depth
  - ⇒ `--isl-schedule-whole-component` disables incremental scheduling
    Incremental scheduling
    - ⋆ disabled by default in `isl`
    - ⋆ enabled by default in `PPCG`

Incremental Scheduling Example

trmm from PolyBench 4.1

```
for (i = 0; i < _PB_M; i++)
   for (j = 0; j < _PB_N; j++) {
      for (k = i+1; k < _PB_M; k++)
         B[i][j] += A[k][i] * B[k][j];
      B[i][j] = alpha * B[i][j];
   }
```

Without incremental scheduling

```
domain: "[n, m] -> { S_3[i, j, k] : i >= 0 and 0 <= j < n and
child:
  schedule: "[n, m] -> [{ S_3[i, j, k] -> [(j)]; S_5[i, j] ->
  permutable: 1
  coincident: [ 1, 0, 0 ]
  child:
    sequence:
    - filter: "[n, m] -> { S_3[i, j, k] }"
    - filter: "[n, m] -> { S_5[i, j] }"
```


Incremental Scheduling Example

trmm from PolyBench 4.1

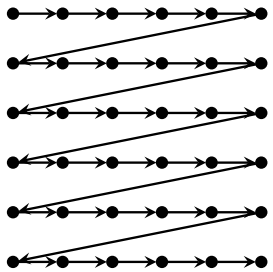
```
for (i = 0; i < _PB_M; i++)
   for (j = 0; j < _PB_N; j++) {
      for (k = i+1; k < _PB_M; k++)
         B[i][j] += A[k][i] * B[k][j];
      B[i][j] = alpha * B[i][j];
   }
```

With incremental scheduling

```
domain: "[n, m] -> { S_3[i, j, k] : i >= 0 and 0 <= j < n and
child:
  sequence:
  - filter: "[n, m] -> { S_3[i, j, k] }"
    child:
      schedule: "[n, m] -> [{ S_3[i, j, k] -> [(j)] }, { S_3[i
      permutable: 1
      coincident: [ 1, 0, 0 ]
  - filter: "[n, m] -> { S_5[i, j] }"
    child:
      schedule: "[n, m] -> [{ S_5[i, j] -> [(i)] }, { S_5[i, j
      permutable: 1
      coincident: [ 1, 1 ]
```


Coalescing

```
for (int i = 0; i < 6; ++i)
  for (int j = 0; j < 6; ++j)
S:  s += f(i, j);
```



Valid schedule: $\{ S[i,j] \to 6i + j \}$

$\Rightarrow$ flattens 2D domain into 1D schedule dimension

$\Rightarrow$ confuses scheduling algorithm

$\Rightarrow$ contains large coefficients

Handling in `isl`:

- Pluto-algorithm (ILP)
  - $\Rightarrow$ impose bounds on coefficients based on instances set sizes
- Feautrier (LP)
  - $\Rightarrow$ detect coalescing in result and retry with smallest coefficient set to zero

Coalescing and Context

Context used sparingly inside PPCG to reduce risk of coalescing

Example:

instance set	$\{ [i,j] : 0 \leqslant i,j < n \}$
	$\Rightarrow$ no risk of coalescing
context	$\{ : n = 1024 \}$
intersection	$\{ [i,j] : 0 \leqslant i,j < 1024 \wedge n = 1024 \}$
	$\Rightarrow$ risk of coalescing
gist	$\{ [i,j] : 0 \leqslant i,j < 1024 \}$
	$\Rightarrow$ risk of coalescing

Note: even simplification (gist) of instance set with respect to context introduces fixed bounds

Use of context can be reconsidered now that coalescing preventing measures have been taken

Proximity Constraints

Recall:

- Proximity $\mathbf{a} \to \mathbf{b}$
  - $\Rightarrow$ statement instance $\mathbf{b}$ preferably executed close to $\mathbf{a}$
  - $\Rightarrow$ $f(\mathbf{b}) - f(\mathbf{a})$ as small as possible

Pluto-algorithm

- looks for uniform bound $f(\mathbf{b}) - f(\mathbf{a}) \leqslant u(\mathbf{n})$ over all such pairs
- "minimizes" $u(\mathbf{n})$

```
A:   a = f1();
     for (int i = 0; i < n; ++i)
B:       A[i] = a;
C:   b = f1();
     for (int i = 0; i < m; ++i)
D:       B[i] = b;
```

Proximity constraints: $\{ \mathtt{A}[] \to \mathtt{B}[i] : 0 \leqslant i < n; \mathtt{C}[] \to \mathtt{D}[i] : 0 \leqslant i < m \}$

- $\Rightarrow$ $u(\mathbf{n})$ needs to be larger than n and m
- $\Rightarrow$ $u(\mathbf{n})$ cannot involve m (constraint $\mathtt{A}[] \to \mathtt{B}[i]$ for every value of m)
- $\Rightarrow$ $u(\mathbf{n})$ cannot involve n (constraint $\mathtt{C}[] \to \mathtt{D}[i]$ for every value of n)
- $\Rightarrow$ no non-trivial solution

Scheduling Issues May 11, 2016 49 / 66

Proximity Constraints

Recall:

- Proximity $\mathbf{a} \to \mathbf{b}$
  - $\Rightarrow$ statement instance $\mathbf{b}$ preferably executed close to $\mathbf{a}$
  - $\Rightarrow$ $f(\mathbf{b}) - f(\mathbf{a})$ as small as possible

Pluto-algorithm

- looks for uniform bound $f(\mathbf{b}) - f(\mathbf{a}) \leqslant u(\mathbf{n})$ over all such pairs
- "minimizes" $u(\mathbf{n})$

Note: if some proximity constraint enforces large $u(\mathbf{n})$
then other proximity constraints are essentially ignored

Scaled Schedules

Feautrier tends to schedule the n statements in an SCC apart

- $\Rightarrow$ $S_i[t, \ldots] \to n\,t + i$
- $\Rightarrow$ carries maximal number of dependences, but
- $\Rightarrow$ introduces large coefficients

By default, `isl` breaks up this pattern into

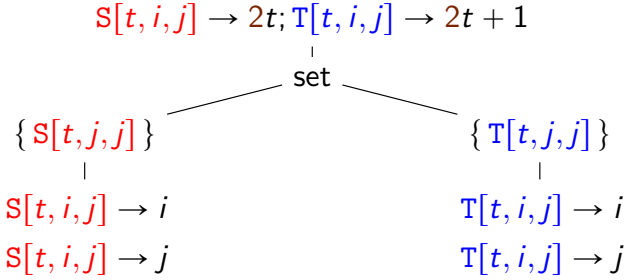
- a scaled down band with the constant terms removed:
  $S_i[t, \ldots] \to t$
- a sequence node ordering statements according to the constant terms
  $\{ S_0[t, \ldots] \}, \{ S_1[t, \ldots] \}, \ldots \{ S_{n-1}[t, \ldots] \},$

Scaled Schedule Example

jacobi-2d from PolyBench 4.1

```
  for (t = 0; t < _PB_TSTEPS; t++) {
      for (i = 1; i < _PB_N - 1; i++)
        for (j = 1; j < _PB_N - 1; j++)
S:        B[i][j] = SCALAR_VAL(0.2) * (A[i][j] + A[i][j-1] +
                                       A[i][1+j] + A[1+i][j] + A[i-1][j]);
      for (i = 1; i < _PB_N - 1; i++)
        for (j = 1; j < _PB_N - 1; j++)
T:        A[i][j] = SCALAR_VAL(0.2) * (B[i][j] + B[i][j-1] +
                                       B[i][1+j] + B[1+i][j] + B[i-1][j]);
  }
```

Pure Feautrier:



Device Mapping [12]

Input: schedule tree

If schedule tree contains no coincident band member

⇒ generate pure CPU code

Otherwise:

- select subtree for mapping to the device
  selected subtree is entire schedule tree, except
  - coincidence-free children of outer set node
  - coincidence-free initial children of outer sequence node
- within selected subtree, generate kernels for
  - outermost bands with coincident members
  - maximal coincidence-free subtrees
    ⇒ insert zero-dimensional band node
- add data copying to/from device around selected subtree
- add device initialization and clean-up around entire schedule tree

Kernel Generation

Input:

- band node with at least one coincident member, or,
- zero-dimensional band node (on top of coincidence-free subtree)

1. Tile entire band
   ⇒ 2 nested bands: "tile band" and "point band"
2. Map outer coincident members (at most 2) of tile band to blocks
3. Map outer coincident members (at most 3) of point band to threads

Motivation for tiling:

- point band has smaller working set
  ⇒ more opportunities for mapping to shared memory
- extra set of coincident band members

Tile, grid and block sizes specified by the user (or some fixed defaults)

- no performance model in PPCG
- band structure depends on scheduler
  ⇒ user typically needs to run PPCG twice

Tiling Example: Matrix Multiplication

```
for (int c0 = 0; c0 < M; c0 += 1)
  for (int c1 = 0; c1 < N; c1 += 1) {
    C[c0][c1] = 0;
    for (int c2 = 0; c2 < K; c2 += 1)
      C[c0][c1] = (C[c0][c1] + (A[c0][c2] * B[c2][c1]));
  }
```

After tiling:

```
for (int c0 = 0; c0 < M; c0 += 32)
  for (int c1 = 0; c1 < N; c1 += 32)
    for (int c2 = 0; c2 < K; c2 += 32)
      for (int c3 = 0; c3 <= ppcg_min(31, M - c0 - 1); c3 += 1)
        for (int c4 = 0; c4 <= ppcg_min(31, N - c1 - 1); c4 += 1) {
          if (c2 == 0)
            C[c0 + c3][c1 + c4] = 0;
          for (int c5 = 0; c5 <= ppcg_min(31, K - c2 - 1); c5 += 1)
            C[c0 + c3][c1 + c4] = (C[c0 + c3][c1 + c4] +
                (A[c0 + c3][c2 + c5] * B[c2 + c5][c1 + c4]));
        }
```

Within point band,

- a single element of C is used per (virtual) thread
- a fixed-size tile of A and B is used

Local Copies

On a CUDA device, shared memory and registers can be accessed more efficiently than global memory

⇒ PPCG tries to copy tiles of arrays to shared memory or registers

Copy is inserted right outside the band mapped to threads

- Data copy operations from/to global memory
- Synchronization to protect local copies

Note:

- Copy may be moved up the tree if this move does not affect the tile
- Data copy operation from/to global memory only added if some data may flow in/out
- Data copy from global memory to shared memory copies entire tile
  ⇒ simpler code
  ⇒ reduced risk of thread divergence

Local Copies Example: Matrix Multiplication

```
for (int c0 = 32 * b0; c0 < M; c0 += 8192)
  for (int c1 = 32 * b1; c1 < N; c1 += 8192) {
    for (int c2 = 0; c2 < K; c2 += 32) {
      if (M >= t0 + c0 + 1)
        for (int c4 = t1; c4 <= ppcg_min(31, K - c2 - 1); c4 += 16)
          shared_A[t0][c4] = A[(t0 + c0) * K + (c2 + c4)];
      if (K >= t0 + c2 + 1)
        for (int c4 = t1; c4 <= ppcg_min(31, N - c1 - 1); c4 += 16)
          shared_B[t0][c4] = B[(t0 + c2) * N + (c1 + c4)];
      barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
      if (M >= t0 + c0 + 1 && N >= t1 + c1 + 1 && c2 == 0) {
        private_C[0][0] = 0;
        if (N >= t1 + c1 + 17)
          private_C[0][1] = 0;
      }
      if (M >= t0 + c0 + 1 && N >= t1 + c1 + 1)
        for (int c3 = 0; c3 <= ppcg_min(31, K - c2 - 1); c3 += 1) {
          private_C[0][0] = (private_C[0][0] + (shared_A[t0][c3] * shared_B[c3][t1]));
          if (N >= t1 + c1 + 17)
            private_C[0][1] = (private_C[0][1] + (shared_A[t0][c3] * shared_B[c3][t1 + 16]));
        }
      barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
    }
    if (M >= t0 + c0 + 1 && N >= t1 + c1 + 1) {
      C[(t0 + c0) * N + (t1 + c1)] = private_C[0][0];
      if (N >= t1 + c1 + 17)
        C[(t0 + c0) * N + (t1 + c1 + 16)] = private_C[0][1];
    }
    barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
  }
```


Array Reference Groups

Multiple copies may be created from different references to the same array

- initially consider each reference separately
- incrementally combine array reference groups that conflict
  - the groups may access the same data
  - at least one reference in one of the groups is a write

Example: `syrk` from PolyBench 4.1

```
for (i = 0; i < _PB_N; i++) {
  for (j = 0; j <= i; j++)
    C[i][j] *= beta;
  for (k = 0; k < _PB_M; k++) {
    for (j = 0; j <= i; j++)
      C[i][j] += alpha * A[i][k] * A[j][k];
  }
}
```

Only one reference to `A` mapped to shared memory

Shared Memory or Registers

No copies are created in the following cases

- read-only scalars (passed as function arguments)
- accesses to slices of an array
- any may-writes that are not also must-writes

A copy to shared memory may be created if

- there is some reuse or access is uncoalesced (relevant for OpenCL?)
- accessed set fits in a rectangular tile with fixed bounds

A copy to registers may be created if

- there is some reuse
- every element is accessed by a single thread
- index expressions only depend on coincident band members
  $\Rightarrow$ band can be sunk and unrolled
- accessed set fits in a rectangular tile with fixed bounds

If both shared memory and registers are possible, then prefer registers

Note: only heuristics, no cost model

Note: some arrays may be forcibly mapped to registers
(temporary arrays in case of live-range reordering)

Device Initialization and Clean-up

Initialization:

- declare device arrays
  - accessed by code mapped to device, and
  - stored in global memory
- initialize device
- allocate device arrays

Clean-up:

- free device arrays
- clean-up device

Data Copying to/from Device

Copy-out:

- take may-writes
- remove writes only needed for dataflow inside selected subtree
- approximate to entire array

May-persist:

- elements that may need to be preserved by selected subtree
- consists of
  - elements that may need to be preserved by entire SCoP
    $\Rightarrow$ elements not definitely written and not definitely killed
  - elements in potential dataflow across selected subtree

May-not-written: $(\text{copy-out} \cap_{\text{ran}} \text{may-persist}) \setminus \text{must-write}$

Copy-in: live-in $\cup$ may-not-written

Note: if array elements are structures, then entire structures are copied

Data Copying Example

```
__pencil_kill(A);
for (int i = 0; i < n; i++)
    if (B[i] > 0)
        A[i] = B[i];
```

`A` may be written

$\Rightarrow$ `A` in copy-out

`A` may also *not* be written (completely), but no data can flow across kill

$\Rightarrow$ ~~parts of `A` may (be expected to) survive~~

$\Rightarrow$ ~~`A` also needs to be in copy-in~~

Potential Future Work for PPCG

- improve efficiency of `isl`
  so far main focus has been on functionality
- dependence analysis in `pet`
- memory compaction/storage optimization
- selective array expansion
- some cost model

References I

[1] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Javed Absar, Sven van Haastregt, Alexey Kravets, Anton Lokhmotov, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Róbert Dávid, and Elnar Hajiyev. "PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming". In: *Proc. Parallel Architectures and Compilation Techniques (PACT'15)*. Oct. 2015. DOI: 10.1109/PACT.2015.17.

[2] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. "Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model". In: *International Conference on Compiler Construction (ETAPS CC)*. Apr. 2008.

References II

[3] Paul Feautrier. "Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time". In: *International Journal of Parallel Programming* 21.6 (Dec. 1992), pp. 389–420. DOI: 10.1007/BF01379404.

[4] Tobias Grosser, Armin Größlinger, and Christian Lengauer. "Polly - Performing polyhedral optimizations on a low-level intermediate representation". In: *Parallel Processing Letters* 22.04 (2012). DOI: 10.1142/S0129626412500107.

[5] Sven Verdoolaege. "isl: An Integer Set Library for the Polyhedral Model". In: *Mathematical Software - ICMS 2010*. Ed. by Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama. Vol. 6327. Lecture Notes in Computer Science. Springer, 2010, pp. 299–302. DOI: 10.1007/978-3-642-15582-6_49.

References III

[6] Sven Verdoolaege. "Counting Affine Calculator and Applications". In: *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*. Chamonix, France, Apr. 2011. DOI: 10.13140/RG.2.1.2959.5601.

[7] Sven Verdoolaege. *PENCIL support in pet and PPCG*. Tech. rep. RT-457, version 2. INRIA Paris-Rocquencourt, May 2015. DOI: 10.13140/RG.2.1.4063.7926.

[8] Sven Verdoolaege. *Presburger Formulas and Polyhedral Compilation*. 2016. DOI: 10.13140/RG.2.1.1174.6323.

[9] Sven Verdoolaege and Albert Cohen. "Live-Range Reordering". In: *Proceedings of the sixth International Workshop on Polyhedral Compilation Techniques*. Prague, Czech Republic, Jan. 2016. DOI: 10.13140/RG.2.1.3272.9680.

References IV

[10] Sven Verdoolaege and Tobias Grosser. "Polyhedral Extraction Tool". In: *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*. Paris, France, Jan. 2012. DOI: 10.13140/RG.2.1.4213.4562.

[11] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. "Schedule Trees". In: *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*. Vienna, Austria, Jan. 2014. DOI: 10.13140/RG.2.1.4475.6001.

[12] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. "Polyhedral parallel code generation for CUDA". In: *ACM Trans. Archit. Code Optim.* 9.4 (2013), p. 54. DOI: 10.1145/2400682.2400713.

References V

[13] Sven Verdoolaege, R. Seghir, K. Beyls, Vincent Loechner, and Maurice Bruynooghe. "Counting integer points in parametric polytopes using Barvinok's rational functions". In: *Algorithmica* 48.1 (June 2007), pp. 37–66. DOI: 10.1007/s00453-006-1231-0.

[14] Doran K. Wilde. *A Library for doing polyhedral operations*. Tech. rep. 785. IRISA, Rennes, France, 1993, 45 p.